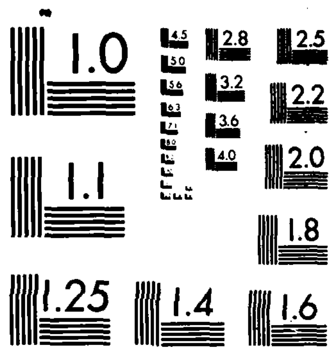
AD-A165 288 ADA (TRADE NAME) TRANING CURRICULUM ADVANCED ADA TOPICS 1/1
L305 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA 1986
DAAB07-83-C-K506

UNCLASSIFIED F/G 5/9 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

1986

Ada® Training Curriculum

AD-A165 288

DTIC FILE COPY

Advanced Ada® Topics
L305
Teacher's Guide
Exercises

DTIC
ELECTE
MAR 1 3 1986
D
E

Superseded
AD-A154 621

86 3 11 142

U.S. Army Communications-Electronics Command
(CECOM)

Contract DAAB07-83-C-K506

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

•Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

*Approved For Public Release/Distribution Unlimited

INSTRUCTORS' GUIDE TO THE L305

There are far more exercises in this booklet than can be solved during one presentation of L305. The instructors should select specific exercises for the class based on the class's mathematical background and programming ability, the pace of the course, and the degree of success solving previous exercises. The purpose of this Guide is to help instructors make this selection.

Each exercise is designed for a certain point in the course. ~~The~~ accompanying diagram shows the point at which each exercise can be assigned. In most cases the exercise makes use of information presented as part of the indicated topic, and should be assigned after that topic has been presented. The exception is exercise 15, which can be solved any time after the generic units topic in Part III, Section 11. We recommend that it be assigned before the sets topic in Part V Section 18 to remind students of the implementation of sets presented at the beginning of Part II Section 3.

Certain of the exercises are based on solutions to other exercises. This is indicated in the diagram by arrows leading from one exercise to another. Exercises 6, 7, and 8 can be solved by editing the solutions to exercises 4, 5, and 3, respectively. It would be helpful for the instructor to install a Square_Root function (parameter and return type Float) in a library unit Math_Package. Exercises 10, 11, and 14 can be solved using the generic package developed as the solution to exercise 8. Exercise 16 is an enhancement of the package developed in exercise 15.

Exercise 1 is strongly recommended. It is a good review of concepts from L202 and a good warm-up. It tests of students' understanding of the fundamental principles underlying packages, particularly the distinction between interface and implementation. The exercise is fairly easy, with one catch. A fixed-point type was deliberately introduced to remind students of the need for type conversion.

Exercises 2 and 3 are both valuable, but both time-consuming. At most one of these exercises should be assigned, or students will quickly grow tired of list manipulation. Exercise 2 supplements the course's brief coverage of double-linked lists. It requires original analytical thought about pointer manipulation. Exercise 3 is a review of singly-linked list manipulation, but also requires students to declare a limited private type. Exercise 3 can be used as a stepping stone to Exercise 8.

Exercises 4 and 5 are provided as alternatives to each other. They are quite similar, and at most one of the two should be assigned. Each requires students to provide a

private type for which arithmetic operations have been overloaded. In each case, a correct solution requires careful consideration of exceptions. Exercise 4 provides a review of complex arithmetic, but previous exposure to complex numbers (at the high school level) would make students more comfortable with this problem. Exercise 5 provides a review of vector arithmetic (addition, multiplication by a scalar, and dot product), but previous exposure is again desirable. Exercise 5 is the only exercise that requires students to write a type declaration with a discriminant.

Exercise 4 can be followed up by exercise 6 and exercise 5 by exercise 7. Both of these follow-up exercises generalize the original solutions by making them generic. The transition from exercise 4 to exercise 6 involves trivial text editing. The transition from exercise 5 to exercise 7 requires more thought, because dimensions of vectors are specified by discriminants in exercise 5 and by a generic formal constant in exercise 7.

Students who have solved exercise 4 will already have done most of the work involved in solving exercise 8. Exercise 8 requires a solid understanding of generic units. If this exercise is not assigned, the package specification should be reviewed in class, since the package is referred to in Part V of the course. Reviewing the specification is sufficient to allow students to use the package in solving exercises 10, 11, and 14.

Exercise 9 is presented as an alternative to exercise 8. Exercise 9 covers material very similar to that presented in Part V Section 14. It is assumed that the generic list package developed in exercise 8 is <u>not</u> used in the solution of exercise 9.

Exercises 10 and 11 are similar to each other, and should not both be assigned. Both show students how the general-purpose list package developed in exercise 8 can be used as a building block to implement a higher-level data abstraction. Both solutions are intricate and time consuming. The two solutions follow almost the same logic, but the addition of polynomials (in exercise 11) is slightly simpler than the addition of natural numbers (in exercise 10) because it does not involve carries. In contrast, the addition of natural numbers is familiar to everybody, but symbolic addition of polynomial formulas is not.

Exercises 12 and 13 are alternatives to each other. Both are simple exercises in tree manipulation and recursion. Exercise 13 uses a type with a discriminant (supplied in the problem statement). It is recommended that one of these exercises be assigned.

Exercise 14 is a modification of a package presented in class. The solution is fairly short. The exercise demonstrates that there can be many implementations of the same data abstraction, with different performance characteristics. (Thus exercise 14 should be described to the class even if it is not assigned.) The solution provides experience in the use of a previously written generic package, and forces students to confront some of the naming problems that can arise when using derived types. (These problems and their solution are described in the derived types topic of Part IV Section 12, but this discussion will seem academic until students encounter the problem themselves.)

Exercise 15 is strongly recommended. It is fairly simple and provides a review of the essential concepts presented in the course -- packages, private types, generics, and overloading. Because of this, it is an appropriate final exercise. The exercise will help Pascal programmers feel more comfortable with Ada by showing them a convenient way to obtain the equivalent of a Pascal set type. Finally, the exercise serves as a good lead-in to Part V Section 18.

Exercise 16 is intended as an extension to exercise 15 for those who finish exercise 15 early or do not find it sufficiently challenging.

The exercises in this booklet are not simple. They are meant not for short in-class drills, but for carefully thought-out solution during one or more lab periods. Close instructor supervision is required to keep students from getting stuck or going astray.

Because L305 emphasizes data abstraction and the construction of software out of components, solutions to most of the exercises are packages rather than subprograms. Thus they are not executable by themselves. It is impossible for us to provide drivers for the packages that students develop. Design of the package specification is a major part of the exercise, but each student's specification may require a different driver. Often students want to write their own drivers, both to test the packages and to enjoy observing the fruits of their labor. This is an admirable attitude, but lab time may be limited. Encourage students not to consider writing drivers before they have successfully compiled the solutions themselves.

Compilation of some of the solutions depends on the compilation of package specifications provided in the problem statement or in an earlier solution. These package statements should be provided online in a public file if possible. Compilation of the solution to exercise 4 requires the compilation of a package named Math_Package providing at least the following function:

```
function Square_Root (X: Float) return Float;
```

Compilation of the solutions to exercises 10, 11, and 14 may depend on the generic package specification developed in exercise 8. If exercise 8 is not assigned, this specification should be made available in a public file. Otherwise, it should be placed in a public file after students have finished working on exercise 8, so that those who did not solve the problem can still work on exercises 10, 11, and 14. Compilation of the solution to exercise 12 requires compilation of the package Binary_Tree_Package given in the exercise. Compilation of the solution to exercise 13 requires compilation of the package Tree_Package given in the exercise.

Accession For
X
A-1

EXERCISE WORKBOOK

VG 846.1

	Section 1:	Review of packages and nonscalar types	1. Package Design
PART I			
	Section 2:	Recursive subprograms	
	Section 3:	Sets using Boolean arrays	
PART II	Section 4:	Linear lists	
	Section 5:	Linked lists and recursive types	2. List Manipulation
	Section 6:	Data type encapsulation	
	Section 7:	Private types	3. Integer List Pkg
PART III	Section 8:	Limited private types	
	Section 9:	Use of exceptions	
	Section 10:	Overloading	4. Complex Number Ops; 5. Vector Ops
PART IV	Section 11:	Generics	6. General Complex; 7. General Vector
	Section 12:	Derived types	
	Section 13:	Unchecked deallocation	
	Section 14:	Generic stacks	9. Generic Queue; 8. Generic List
PART V			10. Large Nos.; 11. Polynomial
	Section 15.	Trees	12. Reverse Binary Tree; 13. Sum Leaves
	Section 16.	Searching	14. Priority Queue
	Section 17.	Sorting	
	Section 18.	Linked list implementation of sets	15. Generic Sets
	Section 19.	Mergeable sets	16. Extend Generic Sets
	Section 20.	Graphs	
	Section 21.	Low-level and implementation-dependent features	
PART VI			
	Section 22.	Example of low-level programming	
PART VII	Section 23.	Overview of Ada tasking	

Exercise 1 - Package Design

Write a package providing the following:

- a type for representing a time of day as a whole number of hours from 0 to 23, a whole number of minutes from 0 to 59, and a whole number of seconds from 0 to 59.

- a type for representing temperatures in the range -50 degrees F to 150 degrees F to within a tenth of a degree.

- a function taking a time of day and returning the approximate temperature at that time on the previous day.

Twenty-five temperature readings, taken every hour on the hour from midnight at the start of the previous day to midnight at the end of the previous day, are stored as a stream of real literals in the file HOURLY.DAT. It should only be necessary to read this file one time. The temperature for a given time of day should be estimated from the hourly readings by linear interpolation: If h0 is the previous hourly reading, h1 is the next hourly reading, and f is the fraction of an hour (between 0.0 and 1.0) that has elapsed since the previous hourly reading, the interpolated estimate is h0 + f * (h1 - h0). To facilitate this computation, you should write a function taking a number of minutes from 0 to 59 and a number of seconds from 0 to 59 and returning the corresponding fraction of an hour as a value in Float range 0.0 .. 1.0.

Exercise 2 - List Manipulation

Write a package providing a list of Integer values, operations for manipulating the list, and exceptions raised by those operations.

The list should be implemented as a doubly-linked list with a dummy list cell.

The operations should include the following:

- a function returning an empty list
- a function indicating whether a given list is empty
- a function taking a list and returning a newly-allocated copy of the list
- two procedures to add specified integers to the list, one at the front and one at the rear
- two procedures to remove integers from a list and place them in an out parameter -- one from the front and one from the rear
- a procedure to insert a new integer in the list just after the first occurrence of another specified integer
- a procedure to remove the first occurrence of a specified integer from the list

You can make these operations easier to implement and save yourself some work by writing three lower-level operations: a procedure to insert a new integer following a given list cell, a procedure to remove a given list cell from the list, and a procedure to search for the first cell in the list containing a specified integer.

It is your responsibility to determine the exceptions that may be raised by the various operations.

Exercise 3 - Integer List Package

Create a package to provide a integer list capability. The package should provide a limited private integer list type and a null (empty) list constant. In order to specify where a given integer is to be stored, a position type should also be specified. A null position constant should also be provided. The default initial value of lists should be the null list. Operations on the list are in terms of lists and positions within the list.

The following capabilities should be provided for integer lists.

- Determine the first position in a list. If the list is empty, the null position should be the result.

- Given a position in a list, determine the next one. If the given position is the last position in the list, the next position is the null position.

- Obtain the value stored in a position.

- Replace the value stored in a position.

- Insert an integer in a list after a given position. If the position is the null position, then store the integer at the front of the list.

- Delete an integer (given its position) from a list.

- Append an integer to the end of a list.

- Determine the length of a list.

- Determine whether two lists have identical contents by using an overloaded equality operator "=".

- Make a copy of a list.

Appropriate exceptions should be raised when needed, e.g., attempting to extract the integer value of the null position, etc.

Exercise 4 - Complex Number Operations

Write a package providing a private type for complex numbers, operations for manipulating complex numbers, and exceptions raised by those operations. The operations include overloaded versions of the operators + (unary and binary), - (unary and binary), abs, *, /, and ** (with a right operand of type Integer). In addition, there should be two functions returning the real part and imaginary part of a complex number as values of type Float and a function taking a real part and an imaginary part as values of type Float and returning the corresponding complex number. Complex operations should raise one exception upon an attempt to divide by 0+0i or raise 0+0i to a negative power and another when computation of the result overflows.

The arithmetic operations on complex numbers work as follows:

```
+(a + bi)             =    a + bi
-(a + bi)             =    -a + (-bi)
(a + bi) + (c + di) =    (a + c) + (b + d)i
(a + bi) - (c + di) =    (a - c) + (b - d)i
(a + bi) * (c + di) =    (ac - bd) + (bc + ad)i
(a + bi) / (c + di) =    (ac + bd)/(c**2 + d**2) + ((bc-ad)/(c**2 + d**2))i
abs (a + bi)          =    Square_Root (a**2 + b**2) -- a value of type Float
```

Exponentiation can be implemented by repeated multiplication. You may assume that the function Square_Root, with a parameter and a result of type Float, is provided by the package Math_Package.

Exercise 5 - Vector Operations

Write a package providing a type for vectors, operations on vectors, and exceptions raised by those operations. The type for vectors should have a discriminant specifying the number of dimensions in the vector. Every declaration of an object in the vector type should be for a vector of some fixed number of dimensions.

Abstractly, a vector of <u>n</u> dimensions can be viewed as a sequence of <u>n</u> values of type Float, (X1, ..., Xn). The operations on vectors are:

- a version of the operator + for adding two vectors with the same number of dimensions to produce a new vector with that number of dimensions:

  (X1, ..., Xn) + (Y1, ..., Yn) = (X1 + Y1, ..., Xn + Yn)

- a version of the operator * for multiplying a left operand of type Float by a right operand of the vector type to produce a new vector:

  X * (Y1, ..., Yn) = (X * Y1, ..., X * Yn)

- a version of the operator * taking two vectors with the same number of dimensions and computing their "dot product" as a value of type Float:

  (X1, ..., Xn) * (Y1, ..., Yn) = X1 * Y1 + ... + Xn * Yn

- a function Zero_Vector taking a Positive parameter and returning a vector with the specified number of dimensions consisting entirely of zeroes:

  Zero_Vector (3) = (0.0, 0.0, 0.0)

- a function Basis_Element taking two parameters of subtype Positive and returning a vector with the number of dimensions indicated by the first parameter, with a value of one in the dimension indicated by the second parameter and a value of zero in all other dimensions:

  Basis_Element (3, 1) = (1.0, 0.0, 0.0)

There should be an exception raised when the computation of a result overflows and another raised when there is a dimension mismatch. (This includes attempts to add vectors with different numbers of dimensions, take the dot product of vectors with different numbers of dimensions, or call Basis_Element with the value of the second parameter exceeding the value of the first parameter.)

Exercise 6 - Generalization of Complex Numbers Package

Redesign the complex numbers package of Exercise 5 as a generic package in which the real and imaginary parts of a complex number are values in some floating-point type specified as a generic parameter.

<u>Write only the generic package declaration, not the package body</u>.

Exercise 7 - Generalization of Vector Package

Redesign the vector package developed in Exercise 6 as a generic package. Allow the role originally played by type Float to be played by any type with operations analogous to the Float operations + and * and values analogous to the Float values 0.0 and 1.0. Also, provide a generic parameter specifying the number of dimensions in the vector type. This generic parameter should have a default value of 3.

Because all vectors in the vector type will have the same number of dimensions, the vector type no longer needs a discriminant. The Zero_Vector function can be replaced by a constant and the first parameter to Basis_Element can be eliminated. Exceptions can no longer arise from number-of-dimensions mismatches.

Write both the generic package declaration and the body.

Exercise 8 - Generic List Package

This exercise expands on Exercise 4. Create a generic list package that provides a limited private list type. The type of the list elements is specified by a generic parameter and may be any non-limited type.

For example, if the generic package is named List_Package_Template, then a list package for a type Type_Of_Interest would be instantiated as

```
package Type_Of_Interest_List_Package is new
        List_Package_Template (Element_Type => Type_Of_Interest);
```

The package should provide suitably modified versions of the types, constants, and operations described in Exercise 4. Also provide a generic procedure that will "process" each element in a list. This procedure should have a generic formal procedure parameter for "processing" a single member of the list. The generic formal procedure should take one <u>in</u> parameter of the list element type, which is the element to be "processed".

Exercise 9 - Generic Queue Package

Write a generic package providing a limited private type for queues, operations on queues, and exceptions. The only generic parameter is the type of the items in the queue.

Use a linked list to implement the queue. Deallocate allocated variables once they are no longer in use.

There should be operations to enqueue an item in the queue (when there is enough storage left to do so), to dequeue an item from a non-empty queue, to determine whether enough space is available to enqueue another item, and to determine whether a queue is empty. You may either make an empty queue the default initial value of all objects in the queue type or else provide another operation to initialize a queue to the empty queue. If you choose the second approach, you should raise an exception upon an attempt to apply any of the other operations to an uninitialized queue.

Exercise 10 - A Package for Very Large Natural Numbers

Write a package providing a limited private type Unbounded_Natural for representing potentially very large natural numbers, along with the following operations:

- an overloaded function taking a parameter of type String containing only digits or a parameter of subtype Natural and returning the corresponding Unbounded_Natural value

- a version of + for Unbounded_Natural

- a version of Put taking a single Unbounded_Natural parameter and printing the corresponding sequence of digits on the standard output file.

Also, provide any exceptions you think are appropriate.

Implement Unbounded_Natural as a linked list of digits. You may use the generic package List_Package_Template handed out as a solution to Exercise 9. You will find it easier to implement + (but slightly harder to implement Put) if you keep this list in reverse order -- that is, with the digit in the one's place at the front of the list and the highest-order digit at the end of the list.

<u>For the ambitious student only</u>: Also provide a version of * for Unbounded_Natural.

Exercise 11 - A Package for Polynomial Formulas

A polynomial is a mathematical formula of the form:

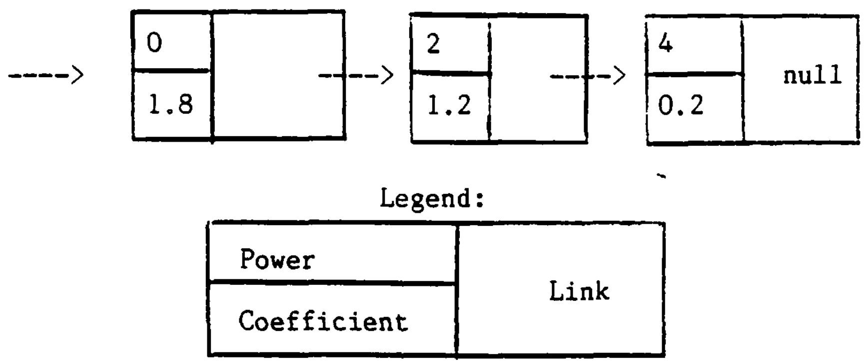
$$C_nX^n + C_{n-1}X^{n-1} + \dots + C_2X^2 + C_1X^1 + C_0X^0$$

Each C_iX^i is called a term of the polynomial and each C_i is called a coefficient. The polynomial is over the variable X.

The polynomials over a given variable can be represented by linked lists in which each list element corresponds to a term. There is a list element for each term with a non-zero coefficient, specifying the value of the coefficient and the exponent of X. (Terms with zero coefficients do not appear in the list.) List elements are arranged in order of ascending powers. For example, the polynomial

$$0.2\ X^4 + 0.0\ X^3 + 1.2\ X^2 + 0.0\ X^1 + 1.8\ X^0$$

(or more simply, $0.2\ X^4 + 1.2\ X^2 + 1.8$) would be represented by a list like the following:



Write a generic package providing a private type for representing polynomial formulas over the variable X. The generic parameters are the type of the coefficients, the coefficient value corresponding to zero, and an operation + defined for the type of the coefficients.

The package should provide:

- a function taking a coefficient C (of the generic formal type) and a power N (of subtype Natural) and returning the one-term polynomial formula CX^N.

- an operator + taking two polynomial formulas and returning the formula for their sum.

- a zero-parameter function returning a polynomial formula in which all coefficients are zero.

Make sure that none of these operations (for example, adding the formula $X^2 + 4X + 4$ to the formula $X^2 - 4X + 4$) creates a list element with a zero coefficient.

You may use the generic package List_Package_Template distributed as a solution to Exercise 9.

<u>For the ambitious student only</u>: Add a generic parameter corresponding to an operation * for the coefficient type, and have the package provide an operation * for the polynomial type. This operation takes two polynomial formulas as operands and returns the formula for their product.

Exercise 12 - Reversing a Binary Tree

The following package provides type declarations for binary trees whose nodes contain pointers to strings:

```
package Binary_Tree_Package is

     type Tree_Node_Type;

     type Tree_Type is access Tree_Node_Type;

     type String_Pointer_Type is access String;

     type Tree_Node_Type is
        record
          Data_Part                          : String_Pointer_Type;
          Left_Child_Part, Right_Child_Part : Tree_Type;
        end record;

end Binary_Tree_Package;
```

Write a function that takes a parameter of type Binary_Tree_Package.Tree_Type and returns the mirror image of the parameter -- another tree in which the left and right subtrees have been exchanged at each level of the tree.

This is a <u>function</u> returning a new tree. The tree passed as a parameter should not be altered.

Exercise 13 - Summing the Leaves of a Tree

The following package provides type declarations for trees in which a node may have any number of children, each leaf contains data of type Integer, and no other node contains data.

```
package Tree_Package is

     type Tree_Node_Type (Number_Of_Children: Natural);

     type Tree_Type is access Tree_Node_Type;

     type Tree_List_Type is array (Positive range <>) of Tree_Type;

     type Tree_Node_Type (Number_Of_Children: Natural) is
        record
          case Number_Of_Children is
             when 0 =>
               Data_Part : Integer;
             when others =>
               Child_List_Part:
                  Tree_List_Type (1 ..  Number_Of_Children);
          end case;
        end record;

end Tree_Package;
```

Write a function taking a parameter of type Tree_Package.Tree_Type and returning the sum of the values at the leaves of the tree.

Exercise 14 - Priority Queue Package

Modify the package Priority_Queue_Package to maintain the list of queue elements in sorted order, with the highest-priority item at the front of the list. This makes Add_Element more complicated and Extract_Element simpler.

Exercise 15 - Generic Set Package

Create a generic package that provides a private type for sets, assuming that the set elements belong to some discrete type. The discrete type is specified by a generic parameter. The set can be implemented as a Boolean array. The set should provide an empty set constant. Exceptions should be provided where necessary.

The package should also provide an unconstrained array type Element_List_Type with components in the discrete type. This type is used as the parameter type of the procedure Set_Of, described below.

The following operations should be provided where A and B are sets, and el, e2, x and y are elements.

SET-VALUED FUNCTIONS

A + B	union
A * B	intersection
A - B	difference
- A	complement
Set_Of((el, ... , en))	create a set with whose members are el, ... , en. (The call on Set_Of shown on the left is with a single aggregate parameter of type Element_List_Type.)
Set_Range(x, y)	create a set with members in the range x .. y.

BOOLEAN-VALUED FUNCTION

Member_Of(x, A)	True if and only if x is a member of A.

PROCEDURES

Extract(A, x)	Remove an arbitrary element from A and place its value in x.
Insert(x, A)	Add the element x to the set A.

Below is an example showing how this set may be used.

```
                -- Using the Set Package

package Character_Set_Package is new Set_Package(Character);

subtype Character_Set_Type is Character_Set_Package.Set_Type;

function "+" (Left, Right: Character_Set_Type) return Character_Set_Type
   renames Character_Set_Package."+";

function Character_Range (Low, High: Character) return Character_Set_Type
   renames Character_Set_Package.Set_Range;

subtype Character_List_Type is Character_Set_Package.Element_List_Type;

function Set_Of (Characters: Character_List_Type) return Character_Set_Type
   renames Character_Set_Package.Set_Of;

function Member_Of (Char: Character; Set: Character_Set_Type) return Boolean
   renames Character_Set_Package.Member_Of;

                          .
                          .
                          .

Letters      : Character_Set_Type :=
                  Character_Range('A', 'Z') + Character_Range('a', 'z');
Digits       : Character_Set_Type := Character_Range('0', '9');
Alphanumerics: Character_Set_Type := Letters + Digits;
Operators    : Character_Set_Type := Set_Of( ('=', '+', '-', '*', '/') );

Next         : Character;
                          .
                          .
                          .

if Member_Of (Next, Set => Letters) then

   --  identifier found
                          .
                          .
                          .

elsif Member_Of (Next, Set => Operators) then

   -- operator found
                          .
                          .
                          .

end if;
```

Exercise 16 - Extending the Generic Set Package

This exercise extends the generic set package from exercise 16. The following operations are to be added.

BOOLEAN-VALUED FUNCTION

A <= B   True if and only if A is a subset of B.

GENERIC SET-VALUED FUNCTION

Provide a generic function Set_Image that maps one set to another by applying its generic function parameter, Element_Image, to each member of the set.

GENERIC SET-PROCESSING PROCEDURE

Provide a generic procedure Process_Each_Element that applies its generic procedure parameter, Process_Element, to each member of a set. Process_Element should take a single <u>in</u> parameter belonging to the type of the set elements.

EXERCISE SOLUTIONS

-- Solution to Exercise 1

```
package Temperature_Package is

   type Time_Of_Day_Type is
      record
         Hours_Part   : Integer range 0 .. 23;
         Minutes_Part : Integer range 0 .. 59;
         Seconds_Part : Integer range 0 .. 59;
      end record;

   type Temperature_Type is delta 0.1 range -50.0 .. 150.0;

   function Temperature_Yesterday
      (Time: Time_Of_Day_Type) return Temperature_Type;

end Temperature_Package;

with Text_IO; use Text_IO;

package body Temperature_Package is

   package Temperature_IO is new Fixed_IO (Temperature_Type); use Temperature_IO;

   Hourly_Reading_Table : array (0 .. 24) of Temperature_Type;
   Data_File            : File_Type;

   subtype Fraction_Type is Float range 0.0 .. 1.0;
   subtype Minutes_Range is Integer range 0 .. 59;

   function Fraction_Of_Hour (Minutes, Seconds : Minutes_Range)
                           return Fraction_Type is

      Seconds_Per_Hour: constant := 3600.0;

   begin   -- Fraction_Of_Hour

      return Float (60 * Minutes + Seconds) / Seconds_Per_Hour;

   end Fraction_Of_Hour;

   function Temperature_Yesterday
      (Time: Time_Of_Day_Type) return Temperature_Type is

      Earlier_Reading : constant Temperature_Type :=
                           Hourly_Reading_Table (Time.Hours_Part);
      Later_Reading   : constant Temperature_Type :=
                           Hourly_Reading_Table (Time.Hours_Part + 1);
      Hour_Fraction   : constant Float :=
                           Fraction_Of_Hour
                              (Time.Minutes_Part, Time.Seconds_Part);
      Prorated_Change : Float;
```

```
   begin  -- Temperature_Yesterday

      Prorated_Change :=
         Hour_Fraction * Float (Later_Reading - Earlier_Reading);
      return Earlier_Reading + Temperature_Type (Prorated_Change);

   end Temperature_Yesterday;


begin  -- Temperature_Package

   Open (Data_File, In_File, "HOURLY.DAT");
   for I in Hourly_Reading_Table'Range loop
      Get (Data_File, Hourly_Reading_Table (I) );
   end loop;
   Close (Data_File);

end Temperature_Package;
```

-- Solution to Exercise 2

```
package Doubly_Linked_List_Package is

   type List_Cell_Type;

   type List_Type is access List_Cell_Type;

   type List_Cell_Type is
      record
         Data_Part          : Integer;
         Forward_Link_Part  : List_Type;
         Backward_Link_Part : List_Type;
      end record;

   function New_Empty_List return List_Type;
   function Is_Empty (List : List_Type) return Boolean;
   function List_Copy (List : List_Type) return List_Type;
   procedure Insert_At_Front (Data : in Integer; List : in List_Type);
   procedure Insert_At_Rear (Data : in Integer; List : in List_Type);
   procedure Insert_Before_Key (Data, Key : in Integer; List : in List_Type);
   procedure Remove_From_Front (Data : out Integer; List : in List_Type);
   procedure Remove_From_Rear (Data : out Integer; List : in List_Type);
   procedure Remove_First_Occurrence (Data : in Integer; List : in List_Type);

   Not_Found_Error, Empty_List_Error : exception;

     -- Not_Found_Error  is raised by Insert_Before_Key when the value  key
     -- is not already in the list and by Remove_First_Occurrence when  the
     -- value data is not already in the list.

     -- Empty_List_Error is raised by Remove_From_Front or Remove_From_Rear
     -- when  called with an empty  list.  Remove_First_Occurrence  raises
     -- Not_Found_Error  rather than Empty_List_Error when called  with  an
     -- empty list.

end Doubly_Linked_List_Package;


package body Doubly_Linked_List_Package is


   procedure Insert_Cell (Data: in Data_Type; Predecessor : in List_Type) is

      Successor : constant List_Type := Predecessor.Forward_Link_Part;
      New_Cell  : constant List_Type :=
                     new List_Cell_Type'
                        (Data_Part          => Data,
                         Forward_Link_Part  => Successor,
                         Backward_Link_Part => Predecessor);
```

```
begin  -- Insert_Cell

   Predecessor.Forward_Link_Part := New_Cell;
   Successor.Backward_Link_Part := New_Cell;

end Insert_Cell;

procedure Remove_Cell (Cell : in List_Type) is

   Predecessor : constant List_Type := Cell.Backward_Link_Part;
   Successor   : constant List_Type := Cell.Forward_Link_Part;

begin  -- Remove_Cell

   Predecessor.Forward_Link_Part := Successor;
   Successor.Backward_Link_Part := Predecessor;

end Remove_Cell;


procedure Search_For_Data
   (Data : in Integer; List : in List_Type; Cell : out List_Type) is

   Next_Cell : List_Type := List.Forward_Link_Part;

begin  -- Search_For_Data

   while Next_Cell /= List loop
      if Next_Cell.Data_Part = Data then
         Cell := Next_Cell;
         return;
      end if;
      Next_Cell := Next_Cell.Forward_Link_Part;
   end loop;
   Cell := null;

end Search_For_Data;


function New_Empty_List return List_Type is

   Result : List_Type := new List_Cell_Type;

begin  -- New_Empty_List

   Result.Forward_Link_Part := Result;
   Result.Backward_Link_Part := Result;
   return Result;

end New_Empty_List;
```

```
function Is_Empty (List : List_Type) return Boolean is
begin
   return List.Forward_Link_Part = List;
end Is_Empty;

function List_Copy (List : List_Type) return List_Type is

   Result: List_Type := New_Empty_List;
     Next_Input_Cell : List_Type := List.Forward_Link_Part;


begin  -- List_Copy

   while Next_Input_Cell /= List loop
      Insert_Cell (Next_Input_Cell.Data_Part, Result.Backward_Link_Part);
      Next_Input_Cell := Next_Input_Cell.Forward_Link_Part;
   end loop;
   return Result;

end List_Copy;


procedure Insert_At_Front (Data : in Integer; List : in List_Type) is
begin
   Insert_Cell (Data, List);
end Insert_At_Front;


procedure Insert_At_Rear (Data : in Integer; Lis t: in List_Type) is
begin
   Insert_Cell (Data, List.Backward_Link_Part);
end Insert_At_Rear;

procedure Insert_Before_Key (Data, Key : in Integer; List : in List_Type) is

   Cell : List_Type;

begin  -- Insert_Before_Key

   Search_For_Data (Key, List, Cell);
   if Cell = null then
      raise Not_Found_Error;
   else
      Insert_Cell (Data, Cell.Backward_Link_Part);
   end if;

end Insert_Before_Key;
```

```
procedure Remove_From_Front (Data : out Integer; List : in List_Type) is

   First_Cell : constant List_Type := List.Forward_Link_Part;

begin  -- Remove_From_Front

   if First_Cell = List then
      raise Empty_List_Error;
   else
      Data := First_Cell.Data_Part;
      Remove_Cell (First_Cell);
   end if;

end Remove_From_Front;


procedure Remove_From_Rear (Data : out Integer; List : in List_Type) is

   Last_Cell : constant List_Type := List.Backward_Link_Part;

begin  -- Remove_From_Rear

   if Last_Cell = List then
      raise Empty_List_Error;
   else
      Data := First_Cell.Data_Part;
      Remove_Cell (Last_Cell);
   end if;

end Remove_From_Rear;


procedure Remove_First_Occurrence (Data : in Integer; List: in List_Type) is

   Cell : List_Type;

begin  -- Remove_First_Occurrence

   Search_For_Data (Data, List, Cell);
   if Cell = null then
      raise Not_Found_Error;
   else
      Remove_Cell (Cell);
   end if;

end Remove_First_Occurrence;

end Doubly_Linked_List_Package;
```

-- Solution to Exercise 3

```
package Integer_List_Package is

   type Integer_List_Type is limited private;
   Null_Integer_List : constant Integer_List_Type;

   type Position_Type is private;
   Null_Position : constant Position_Type;

   Position_Error : exception;

   function First_Position
             (Integer_List : Integer_List_Type) return Position_Type;

   function Next_Position (Position : Position_Type) return Position_Type;

   function Integer_Value (Position : Position_Type) return Integer;

   procedure Replace_Integer
                (Position : in out Position_Type;
                 Element  : in Integer);

   procedure Insert_Integer
                (Integer_List : in out Integer_List_Type;
                 Element      : in Integer;
                 After        : in Position_Type);

   procedure Delete_Integer
                (Integer_List : in out Integer_List_Type;
                 Position     : in Position_Type);

   procedure Append_Integer
                (Integer_List : in out Integer_List_Type;
                 Element      : in Integer) is

   function Length (Integer_List : Integer_List_Type) return Natural;

   function "=" (Left, Right : Integer_List_Type) return Boolean;

   procedure Copy_Integer_List
                (From : in Integer_List_Type;
                 To   : out Integer_List_Type);

private

   type Integer_List_Cell_Type;
   type Position_Type is access Integer_List_Cell_Type;
```

```
   type Integer_List_Cell_Type is
      record
         Integer_Part : Integer;
         Link_Part    : Position_Type;
      end record;


   Null_Position : constant Position_Type := null;

   type Integer_List_Type is
      record
         Length_Part         : Natural := 0;
         First_Position_Part : Position_Type := Null_Position;
         Last_Position_Part  : Position_Type := Null_Position;
      end record;

   Null_Integer_List : constant Integer_List_Type :=
                          (0, Null_Position, Null_Position);

end Integer_List_Package;


package body Integer_List_Package is

   function First_Position
            (Integer_List : Integer_List_Type) return Position_Type is
   begin

      return Integer_List.First_Position_Part;

   end First_Position;

   function Next_Position (Position : Position_Type) return Position_Type is
   begin

     if Position = Null_Position then
        raise Position_Error;
     else
        return Position.Link_Part;
     end if;

   end Next_Position;

   function Integer_Value (Position : Position_Type) return Integer is
   begin

      if Position = Null_Position then
         raise Position_Error;
      else
         return Position.Integer_Part;
      end if;

   end Integer_Value;
```

```
procedure Replace_Integer
          (Position : in out Position_Type;
           Element  : in Integer) is

begin

   if Position = Null_Position then
      raise Position_Error;
   else
      Position.Integer_Part := Element;
   end if;

end Replace_Integer;

procedure Insert_Integer
          (Integer_List : in out Integer_List_Type;
           Element      : in Integer;
           After        : in Position_Type) is
begin

   if After = Null_Position then  -- insert at front

      declare

         New_Position : Position_Type :=
            new Integer_List_Cell_Type'(Element,
            Integer_List.First_Position_Part);
      begin

         Integer_List.First_Position_Part := New_Position;
         if Integer_List.Length_Part = 0 then
            Integer_List.Last_Position_Part := New_Position;
         end if;
         Integer_List.Length_Part := Integer_List.Length_Part + 1;

      end;  -- block

   else

      declare

         Position         : Position_Type renames After;
         Current_Position : Position_Type := Integer_List.First_Position_Part;
      begin

         --  Search for Position

         while Current_Position /= Null_Position and
                  Current_Position /= Position loop
            Current_Position := Current_Position.Link_Part;
         end loop;

         --  If the position was not found then raise an exception;
         --     otherwise, add the element.
```

```
            if Current_Position /= Position then  -- Position not found
               raise Position_Error;
            else
               Position.Link_Part :=
                  new Integer_List_Cell_Type'(Element, Position.Link_Part);
               if Position = Integer_List.Last_Position_Part then
                  Integer_List.Last_Position_Part := Position.Link_Part;
               end if;
            end if;
         end;  -- block
      end if;
   end Insert_Integer;

   procedure Delete_Integer
                 (Integer_List : in out Integer_List_Type;
                  Position     : in Position_Type) is
   begin

      if (Position = Null_Position) or (Integer_List.Length_Part = 0) then
         raise Position_Error;
      else
         declare

            Previous_Position : Position_Type := Null_Position;
            Current_Position  : Position_Type := Integer_List.First_Position_Part;

         begin

            --  Search for Position

            while Current_Position /= Null_Position and
                  Current_Position /= Position loop
               Previous_Position := Current_Position;
               Current_Position := Current_Position.Link_Part;
            end loop;

            --  If the Position was not found, then raise an exception;
            --     otherwise delete the element.

            if Current_Position /= Position then  -- Position not found
               raise Position_Error;
            else
               if Integer_List.Last_Position_Part = Position then
                  Integer_List.Last_Position_Part := Previous_Position;
               end if;
               if Integer_List.First_Position_Part = Position then
                  Integer_List.First_Position_Part := Position.Link_Part;
               else
                  Previous_Position.Link_Part := Position.Link_Part;
               end if;
               Integer_List.Length_Part := Integer_List.Length_Part - 1;
            end if;
         end; -- block;
      end if;
   end Delete_Integer;
```

```
procedure Append_Integer
            (Integer_List : in out Integer_List_Type;
             Element      : in Integer) is

   Position : Position_Type :=
                 new Integer_List_Cell_Type'(Element, Null_Position);

begin

   if Integer_List.Length_Part = 0 then
      Integer_List.First_Position_Part := Position;
   else
      Integer_List.Last_Position_Part.Link_Part := Position;
   end if;
   Integer_List.Last_Position := Position;
   Integer_List.Length_Part := Length_Part + 1;

end Append_Integer;

function Length (Integer_List : Integer_List_Type) return Natural is
begin

   return Integer_List.Length_Part;

end Length;

function "=" (Left, Right : Integer_List_Type) return Boolean is
begin

   if Left.Length_Part /= Right.Length_Part then
      return False;
   else
      declare

         Left_Position  : Position_Type := Left.First_Position_Part;
         Right_Position : Position_Type := Right.First_Position_Part;

      begin

         while Left_Position /= Null_Position loop
            if Left_Position.Integer_Part = Right_Position.Integer_Part then
               Left_Position := Left_Position.Link_Part;
               Right_Position := Right_Position.Link_Part;
            else
               return False;
            end if;
         end loop;

         return True;

      end;  -- block
   end if;
end "=";
```

```
   procedure Copy_Integer_List
                (From : in Integer_List_Type;
                 To   : out Integer_List_Type) is
   begin

      if From.Length_Part = 0 then
         To := Null_Integer_List;
      else

         declare

            From_Position    : Position_Type := From.First_Position_Part;
            Position         : Position_Type;
            New_Integer_List : Integer_List_Type;

         begin

            Position := new Integer_List_Cell_Type'
                               (From_Position.Integer_Part, Null_Position);
            New_Integer_List.Length_Part := From.Length_Part;
            New_Integer_List.First_Position_Part := Position;
            while From_Position.Link_Part /= Null_Position loop
               From_Position := From_Position.Link_Part;
               Position.Link_Part := new Integer_List_Cell_Type'
                                        (From_Position.Integer_Part,
                                         Null_Position);
               Position := Position.Link_Part;
            end loop;
            New_Integer_List.Last_Position_Part := Position;
            To := New_Integer_List;

          end;  --  block

       end if;

   end Copy_Integer_List;

end Integer_List_Package;
```

-- Solution to Exercise 4

```
package Math_Package is
   -- ...
   function Square_Root (x: Float) return Float;
   -- ...
end Math_Package;


package Complex_Number_Package is

   type Complex_Number_Type is private;

   function "+" (Right : Complex_Number_Type) return Complex_Number_Type;
   function "-" (Right : Complex_Number_Type) return Complex_Number_Type;
   function "abs" (Right : Complex_Number_Type) return Float;

   function "+" (Left, Right : Complex_Number_Type) return Complex_Number_Type;
   function "-" (Left, Right : Complex_Number_Type) return Complex_Number_Type;
   function "*" (Left, Right : Complex_Number_Type) return Complex_Number_Type;
   function "/" (Left, Right : Complex_Number_Type) return Complex_Number_Type;

   function "**"
      (Left : Complex_Number_Type; Right : Integer) return Complex_Number_Type;

   function Real_Part (Complex_Number : Complex_Number_Type) return Float;
   function Imaginary_Part (Complex_Number : Complex_Number_Type) return Float;
   function New_Complex_Number
      (Real_Part, Imaginary_Part : Float) return Complex_Number_Type;

   Complex_Division_Error, Complex_Overflow : exception;

private

   type Complex_Number_Type is
      record
         Real_Part, Imaginary_Part : Float;
      end record;

end Complex_Number_Package;
```

```
with Math_Package;

package body Complex_Number_Package is

   function "+" (Right : Complex_Number_Type) return Complex_Number_Type is
   begin
      return Right;
   end "+";

   function "-" (Right : Complex_Number_Type) return Complex_Number_Type is
   begin
      return (-Right.Real_Part, -Right.Imaginary_Part);
   end "-";

   function "abs" (Right : Complex_Number_Type) return Float is

   begin  -- "abs"

      return Math_Package.Square_Root
                (Right.Real_Part ** 2 + Right.Imaginary_Part ** 2);

   exception  -- "abs"

      when Numeric_Error => raise Complex_Overflow;

   end "abs";

   function "+"
      (Left, Right : Complex_Number_Type) return Complex_Number_Type is

   begin  -- "+"

      return (Left.Real_Part + Right.Real_Part,
              Left.Imaginary_Part + Right.Imaginary_Part);

   exception  -- "+"

      when Numeric_Error => raise Complex_Overflow;

   end "+";

   function "-"
      (Left, Right : Complex_Number_Type) return Complex_Number_Type is
```

```
begin  -- "-"

   return (Left.Real_Part - Right.Real_Part,
           Left.Imaginary_Part - Right.Imaginary_Part);

exception  -- "-"

   when Numeric_Error => raise Complex_Overflow;

end "-";

function "*"
   (Left, Right : Complex_Number_Type) return Complex_Number_Type is

begin  -- "*"

   return (Left.Real_Part * Right.Real_Part -
              Left.Imaginary_Part * Right.Imaginary_Part,
           Left.Imaginary_Part * Right.Real_Part +
              Left.Real_Part + Right.Imaginary_Part);

exception  -- "*"

   when Numeric_Error => raise Complex_Overflow;

end "*";


function "/"
   (Left, Right : Complex_Number_Type) return Complex_Number_Type is

   Divisor, Result_Real_Part, Result_Imaginary_Part : Float;

begin  -- "/"

   Divisor := Right.Real_Part ** 2 + Right.Imaginary_Part ** 2;

   if Divisor = 0.0 then
      raise Complex_Division_Error;
   else
      Result_Real_Part :=
         (Left.Real_Part * Right.Real_Part +
          Left.Imaginary_Part * Right.Imaginary_Part) / Divisor;
      Result_Imaginary_Part :=
         (Left.Imaginary_Part * Right.Real_Part -
          Left.Real_Part * Right.Imaginary_Part) / Divisor;
      return (Result_Real_Part, Result_Imaginary_Part);
   end if;

exception  -- "/"

   when Numeric_Error => raise Complex_Overflow;

end "/";
```

```
function "**"
   (Left : Complex_Number_Type; Right : Integer)
   return Complex_Number_Type is

   Zero              : constant Complex_Number_Type := (0.0, 0.0);
   One               : constant Complex_Number_Type := (1.0, 0.0);
   Product, Inverse : Complex_Number_Type;
   Divisor           : Float;

begin  -- "**"

   if Right >= 0 then

      Product := One;
      for K in 1 .. Right loop
         Product := Product * Left;
      end loop;
      return Product;

      -- To be consistent with Ada rules for real exponentiation,
      --   Zero ** 0 returns One.  (Mathematically, the result is
      --   undefined.)

   else

      if Left = Zero then
         raise Complex_Division_Error;
      else
         Inverse := Left ** (-Right);  -- recursive call
         Divisor := Inverse.Real_Part ** 2 + Inverse.Imaginary_Part ** 2;
         return (Inverse.Real_Part / Divisor,
                 -Inverse.Imaginary_Part / Divisor);
            -- equivalent to complex quotient One / Inverse
      end if;

   end if;

exception  -- "**"

   when Numeric_Error => raise Complex_Overflow;

end "**";


function Real_Part (Complex_Number : Complex_Number_Type) return Float is
begin
   return Complex_Number.Real_Part;
end Real_Part;
```

```
   function Imaginary_Part
      (Complex_Number : Complex_Number_Type) return Float is
   begin
      return Complex_Number.Imaginary_Part;
   end Imaginary_Part;


   function New_Complex_Number
      (Real_Part, Imaginary_Part : Float) return Complex_Number_Type is
   begin
      return (Real_Part, Imaginary_Part);
   end New_Complex_Number;

end Complex_Number_Package;
```

-- Solution to Exercise 5

```
package Vector_Package

   type Vector_Type (Number_Of_Dimensions : Positive) is private;

   function "+" (Left, Right : Vector_Type) return Vector_Type;
   function "*" (Left : Float; Right : Vector_Type) return Vector_Type;
   function "*" (Left, Right : Vector_Type) return Float;
   function Zero_Vector
      (Number_Of_Dimensions : Positive) return Vector_Type;
   function Basis_Element
      (Number_Of_Dimensions, Unit_Length_Dimension : Positive)
      return Vector_Type;

   Vector_Overflow, Dimension_Mismatch : exception;

private

   type Component_List_Type is array (Positive range <>) of Float;

   type Vector_Type (Number_Of_Dimensions : Positive) is
      record
         Component_List_Part :
            Component_List_Type (1 .. Number_Of_Dimensions);
      end record;

end Vector_Package;


package body Vector_Package is


   function "+" (Left, Right : Vector_Type) return Vector_Type is

      Result : Vector_Type (Left.Number_Of_Dimensions);

   begin  -- "+"

      if Left.Number_Of_Dimensions /= Right.Number_Of_Dimensions then
         raise Dimension_Mismatch;
      else
         for I in Result.Component_List_Part'Range loop
            Result.Component_List_Part (I) :=
               Left.Component_List_Part (I) + Right.Component_List_Part (I);
         end loop;
         return Result;
      end if;
```

```
exception  -- "+"

   when Numeric_Error => raise Vector_Overflow;

end "+";


function "*" (Left : Float; Right : Vector_Type) return Vector_Type is

   Result : Vector_Type (Right.Number_Of_Dimensions);

begin  -- "*" (Left : Float; Right : Vector_Type) return Vector_Type

   for I in Right.Component_List_Part'Range loop
      Result.Component_List_Part (I) :=
         Left * Right.Component_List_Part (I);
   end loop;
   return Result;

exception  -- "*" (Left : Float; Right : Vector_Type) return Vector_Type

   when Numeric_Error => raise Vector_Overflow;

end "*" ;  -- (Left : Float; Right : Vector_Type) return Vector_Type


function "*" (Left, Right : Vector_Type) return Float is

   Result : Float := 0.0;

begin  -- "*" (Left, Right : Vector_Type) return Float

   if Left.Number_Of_Dimensions /= Right.Number_Of_Dimensions then
      raise Dimension_Mismatch;
   else
      for I in Left.Component_List_Part'Range loop
         Result :=
            Result +
               Left.Component_List_Part (I) *
               Right.Component_List_Part (I);
      end loop;
      return Result;
   end if;

exception  -- "*" (Left, Right : Vector_Type) return Float

   when Numeric_Error => raise Vector_Overflow;

end "*";  -- (Left, Right : Vector_Type) return Float

-- To maximize numerical accuracy, the algorithm for dot product should be
-- modified.  The terms should be summed using double precision for the
-- intermediate terms.  Positive and negative terms should be summed separately,
-- and terms with the smallest absolute value should be accumulated first.
```

```
function Zero_Vector
   (Number_Of_Dimensions : Positive) return Vector_Type is
begin
   return (Number_Of_Dimensions, (1 .. Number_Of_Dimensions => 0.0));
end Zero_Vector;


function Basis_Element
   (Number_Of_Dimensions, Unit_Length_Dimension : Positive)
   return Vector_Type is

   Result : Vector_Type (Number_Of_Dimensions);

begin  -- Basis_Element

   if Unit_Length_Dimension in 1 .. Number_Of_Dimensions then
      Result.Component_List_Part := (1 .. Number_Of_Dimensions => 0.0);
      Result.Component_List_Part (Unit_Length_Dimension) := 1.0;
      return Result;
   else
      raise Dimension_Mismatch;
   end if;

end Basis_Element;


end Vector_Package;
```

-- Solution to Exercise 6

```
generic

   type Real_Type is digits <>;

package Complex_Number_Package is

   type Complex_Number_Type is private;

   function "+" (Right : Complex_Number_Type) return Complex_Number_Type;
   function "-" (Right : Complex_Number_Type) return Complex_Number_Type;
   function "abs" (Right : Complex_Number_Type) return Real_Type;

   function "+" (Left, Right : Complex_Number_Type) return Complex_Number_Type;
   function "-" (Left, Right : Complex_Number_Type) return Complex_Number_Type;
   function "*" (Left, Right : Complex_Number_Type) return Complex_Number_Type;
   function "/" (Left, Right : Complex_Number_Type) return Complex_Number_Type;

   function "**"
      (Left : Complex_Number_Type; Right : Integer)
      return Complex_Number_Type;

   function Real_Part (Complex_Number : Complex_Number_Type) return Real_Type;
   function Imaginary_Part
      (Complex_Number : Complex_Number_Type) return Real_Type;
   function New_Complex_Number
      (Real_Part, Imaginary_Part : Real_Type) return Complex_Number_Type;

   Complex_Division_Error, Complex_Overflow : exception;

private

   type Complex_Number_Type is
      record
         Real_Part, Imaginary_Part : Real_Type;
      end record;

end Complex_Number_Package;
```

-- Solution to Exercise 7

```
generic

   Number_Of_Dimensions : in Positive := 3;
   type Scalar_Type is private;
   Zero, One : in Scalar_Type;
   with function "+" (Left, Right : Scalar_Type) return Scalar_Type is <>;
   with function "*" (Left, Right : Scalar_Type) return Scalar_Type is <>;

package Vector_Package

   type Vector_Type is private;
   subtype Dimension_Subtype is Integer range 1 .. Number_Of_Dimensions;

   function "+" (Left, Right : Vector_Type) return Vector_Type;
   function "*" (Left : Scalar_Type; Right : Vector_Type) return Vector_Type;
   function "*" (Left, Right : Vector_Type) return Scalar_Type;
   function Basis_Element (Dimension : Dimension_Subtype) return Vector_Type;

   Zero_Vector : constant Vector_Type;

   Scalar_Operation_Error : exception;

private

   type Vector_Type is array (1 .. Number_Of_Dimensions) of Scalar_Type;

   Zero_Vector : constant Vector_Type := (1 .. Number_Of_Dimensions => Zero);

end Vector_Package;


package body Vector_Package is


   Basis_Element_List : array (1 .. Number_Of_Dimensions) of Vector_Type;


   function "+" (Left, Right : Vector_Type) return Vector_Type is

      Result : Vector_Type;

   begin  -- "+"

      for I in Vector_Type'Range loop
         Result (I) := Left (I) + Right (I);
      end loop;
      return Result;
```

```
   exception  -- "+"

      when others => raise Scalar_Operation_Error;

   end "+";

   function "*" (Left : Scalar_Type; Right : Vector_Type) return Vector_Type is

      Result : Vector_Type;

   begin  -- "*" (Left : Scalar_Type; Right : Vector_Type) return Vector_Type

      for I in Vector_Type'Range loop
         Result (I) := Left * Right (I);
      end loop;
      return Result;

   exception  -- "*" (Left : Scalar_Type; Right : Vector_Type) return Vector_Type

      when others => raise Scalar_Operation_Error;

   end "*" ;  -- (Left : Scalar_Type; Right : Vector_Type) return Vector_Type


   function "*" (Left, Right : Vector_Type) return Scalar_Type is

      Result : Scalar_Type := Zero;

   begin  -- "*" (Left, Right : Vector_Type) return Scalar_Type

      for I in Vector_Type'Range loop
         Result := Result + Left (I) * Right (I);
      end loop;
      return Result;

   exception  -- "*" (Left, Right : Vector_Type) return Scalar_Type

      when others => raise Scalar_Operation_Error;

   end "*";  -- (Left, Right : Vector_Type) return Scalar_Type

   function Basis_Element (Dimension : Dimension_Subtype) return Vector_Type is
   begin
      return Basis_Element_List (Dimension);
   end Basis_Element;


begin  -- Vector_Package

   for I in Vector_Type'Range loop
      Basis_Element_List (I) := Zero_Vector;
      Basis_Element_List (I) (I) := One;
   end loop;

end Vector_Package;
```

```
                    -- Solution to Exercise 8

generic

   type Element_Type private;

package List_Package_Template is

   type List_Type is limited private;
   Null_List : constant List_Type;

   type Position_Type is private;
   Null_Position : constant Position_Type;

   Position_Error : exception;

   function First_Position(List : List_Type) return Position_Type;

   function Next_Position(Position : Position_Type) return Position_Type;

   function Element_Value(Position : Position_Type) return Element_Type;

   procedure Replace_Element
                (Position : in out Position_Type;
                 Element  : in Element_Type);

   procedure Insert_Element
                (List    : in out List_Type;
                 Element : in Element_Type;
                 After   : in Position_Type);

   procedure Delete_Element
                (List     : in out List_Type;
                 Position : in Position_Type);

   procedure Append_Element
                (List     : in out List_Type;
                 Element  : in Element_Type);

   function Length(List : List_Type) return Natural;

   function "=" (Left, Right : List_Type) return Boolean;

   procedure Copy_List(From : in List_Type; To : out List_Type);

   generic
      with procedure Process_Element(Element : in out Element_Type);
   procedure Process_Each_Element(List : in List_Type);
```

```
private

   type List_Cell_Type;
   type Position_Type is access List_Cell_Type;

   type List_Cell_Type is
      record
         Element_Part : Element_Type;
         Link_Part    : Position_Type;
      end record;

   Null_Position : constant Position_Type  := null;

   type List_Type is
      record
         Length_Part        : Natural   := 0;
         Chain_Part         : Position_Type := Null_Position;
         Last_Position_Part : Position_Type := Null_Position;
      end record;

   Null_List : constant List_Type := (0, Null_Position, Null_Position);

end List_Package_Template;


package body List_Package_Template is

   function First_Position(List : List_Type) return Position_Type is
   begin

      return List.Chain_Part;

   end First_Position;

   function Next_Position(Position : Position_Type) return Position_Type is
   begin

     if Position = Null_Position then
        raise Position_Error;
     else
        return Position.Link_Part;
     end if;

   end Next_Position;

   function Element_Value(Position : Position_Type) return Element_Type is
   begin

      if Position = Null_Position then
         raise Position_Error;
      else
         return Position.Element_Part;
      end if;
```

```
end Element_Value;

procedure Replace_Element
             (Position : in out Position_Type;
              Element  : in Element_Type);
begin

   if Position = Null_Position then
      raise Position_Error;
   else
      Position.Element_Part := Element;
   end if;

end Replace_Element;

procedure Insert_Element
             (List    : in out List_Type;
              Element : in Element_Type;
              After   : in Position_Type) is
begin

   if After = Null_Position then  -- insert at front

      declare

         New_Position : Position_Type :=
                           new List_Cell_Type(Element, List.Chain_Part);

      begin

         List.Chain_Part := New_Position;
         if List.Length_Part = 0 then
            List.Last_Position_Part := New_Position;
         end if;
         List.Length_Part := List.Length_Part + 1;

      end;  -- block

   else

      declare

         Position         : Position_Type renames After;
         Current_Position : Position_Type := List.Chain_Part;

      begin

         --  Search for Position

         while Current_Position /= Null_Position and
                  Current_Position /= Position loop
            Current_Position := Current_Position.Link_Part;
         end loop;
```

```
            --  If the position was not found then raise an exception;
            --     otherwise, add the element.

            if Current_Position /= Position then  -- Position not found
               raise Position_Error;
            else
               Position.Link_Part :=
                  new List_Cell_Type'(Element, Position.Link_Part);
              if Position = List.Last_Position_Part then
                  List.Last_Position_Part := Position.Link_Part;
               end if;
            end if;

         end;  -- block

      end if;

   end Insert_Element;

   procedure Delete_Element
                (List     : in out List_Type;
                 Position : in Position_Type) is
   begin

      if (Position = Null_Position) or (List.Length_Part = 0) then
         raise Position_Error;
      else
         declare

            Previous_Position : Position_Type := Null_Position;
            Current_Position  : Position_Type := List.Chain_Part;

         begin

            --  Search for Position

            while Current_Position /= Null_Position and
                     Current_Position /= Position loop
               Previous_Position := Current_Position;
               Current_Position := Current_Position.Link_Part;
            end loop;

            --  If the Position was not found, then raise an exception;
            --     otherwise delete the element.

            if Current_Position /= Position then  -- Position not found
               raise Position_Error;
            else
               if List.Last_Position_Part = Position then
                  List.Last_Position_Part := Previous_Position;
               end if;
```

```
            if List.Chain_Part = Position then
               List.Chain_Part := Position.Link_Part;
            else
               Previous_Position.Link_Part := Position.Link_Part;
            end if;
            List.Length_Part := List.Length_Part - 1;
         end if;

      end; -- block;

   end if;

end Delete_Element;

procedure Append_Element
             (List     : in out List_Type;
              Element  : in Element_Type) is

   Position : Position_Type := new List_Cell_Type'(Element, Null_Position);

begin

   if List.Length_Part = 0 then
      List.Chain_Part := Position;
   else
      List.Last_Position_Part.Link_Part := Position;
   end if;
   List.Last_Position := Position;
   List.Length_Part := Length_Part + 1;

end Append_Element;

function Length(List : List_Type) return Natural is
begin

   return List.Length_Part;

end Length;

function "=" (Left, Right : List_Type) return Boolean is
begin

   if Left.Length_Part /= Right.Length_Part then
      return False;
   else
      declare

         Left_Position  : Position_Type := Left.Chain_Part;
         Right_Position : Position_Type := Right.Chain_Part;
```

```
      begin

         while Left_Position /= Null_Position loop
            if Left_Position.Element_Part = Right_Position.Element_Part then
               Left_Position := Left_Position.Link_Part;
               Right_Position := Right_Position.Link_Part;
            else
               return False;
            end if;
         end loop;

         return True;

      end;  -- block

   end if;

end "=";

procedure Copy_List(From : in List_Type; To : out List_Type) is
begin

   if From.Length_Part = 0 then
      To := Null_List;
   else

      declare

         From_Position : Position_Type := From.Chain_Part;
         Position : Position_Type;
         New_List : List_Type;

      begin

         Position := new List_Cell_Type'
                           (From_Position.Element_Part, Null_Position);
         New_List.Length_Part := From.Length_Part;
         New_List.Chain_Part := Position;
         while Position_Part.Link_Part /= Null_Position loop
            Position_Part := Position_Part.Link_Part;
            Position.Link_Part := new List_Cell_Type'
                                    (Position_Part.Element_Part,
                                     Null_Position);
            Position := Position.Link_Part;
         end loop;
         New_List.Last_Position_Part := Position;
         To := New_List;

       end;  --  block

    end if;

end Copy_List;
```

```
   procedure Process_Each_Element(List : in List_Type) is

      Position: Position_Type := List.Chain_Part;

   begin

      while Position /= Null_Position loop
         Process_Element(Position.Element_Part);
         Position := Position.Link_Part;
      end loop;

   end Process_Each_Element;

end List_Package_Template;
```

```
generic

   type Element_Type is private;

package Queue_Package_Template is

   type Queue_Type is limited private;

   procedure Initialize_Queue (Queue : out Queue_Type);
   procedure Enqueue (Queue : in out Queue_Type; Element : in Element_Type);
   procedure Dequeue (Queue : in out Queue_Type; Element : out Element_Type);
   function Is_Empty (Queue : Queue_Type) return Boolean;
   function Queue_Space_Available return Boolean;

   Queue_Initialization_Error, Empty_Queue_Error, Queue_Space_Error :
      exception;

private

   type List_Cell_Type;

   type List_Cell_Pointer_Type is access List_Cell_Type;

   type List_Cell_Type is
      record
         Element_Part : Element_Type;
         Link_Part    : List_Cell_Pointer_Type;
      end record;

   type Queue_Type is
      record
         Front_Part, Back_Part : List_Cell_Pointer_Type;
      end record;

end Queue_Package_Template;


package body Queue_Package_Template is


   package Allocation_Package is
      function New_Cell return List_Cell_Pointer_Type;
         -- not to be called when Out_Of_Storage is true
      procedure Recycle_Cell (Cell_Pointer : in out List_Cell_Pointer_Type);
      function Out_Of_Storage return Boolean;
   end Allocation_Package;


   package body Allocation_Package is separate;
```

```
procedure Initialize_Queue (Queue : out Queue_Type) is

   Dummy_Cell : List_Cell_Pointer_Type;

begin  -- Initialize

   if Allocation_Package.Out_Of_Storage then
      raise Queue_Space_Error;
   else
      Dummy_Cell := Allocation_Package.New_Cell;
      Queue := (Front_Part | Back_Part => Dummy_Cell);
   end if;

end Initialize_Queue;


procedure Enqueue (Queue : in out Queue_Type; Element : in Element_Type) is

   Back_Cell_Pointer : List_Cell_Pointer_Type renames Queue.Back_Part;

begin  -- Enqueue

   if Back_Cell_Pointer = null then
      raise Queue_Initialization_Error;
   elsif Allocation_Package.Out_Of_Storage then
      raise Queue_Space_Error;
   else
      Back_Cell_Pointer.Link_Part := Allocation_Package.New_Cell;
      Back_Cell_Pointer := Back_Cell_Pointer.Link_Part;
      Back_Cell_Pointer.Element_Part := Element;
   end if;

end Enqueue;


procedure Dequeue
          (Queue : in out Queue_Type; Element : out Element_Type) is

   Front_Cell_Pointer : List_Cell_Pointer_Type renames Queue.Front_Part;
   Old_Cell           : List_Cell_Pointer_Type;

begin  -- Dequeue

   if Front_Cell_Pointer = null then
      raise Queue_Initialization_Error;
   elsif Front_Cell_Pointer = Queue.Back_Part then
      raise Empty_Queue_Error;
   else
      Element := Front_Cell_Pointer.Element_Part;
      Old_Cell := Front_Cell_Pointer;
      Front_Cell_Pointer := Front_Cell_Pointer.Link_Part;
      Allocation_Package.Recycle_Cell (Old_Cell);
   end if;

end Dequeue;
```

```
   function Is_Empty (Queue : Queue_Type) return Boolean is

   begin  -- Is_Empty

      if Queue.Front_Part = null then
         raise Queue_Initialization_Error;
      else
         return Queue.Front_Part = Queue.Back_Part;
      end if;

   end Is_Empty;

   function Queue_Space_Available return Boolean is

   begin  -- Queue_Space_Available

      return not Allocation_Package.Out_Of_Storage;

   end Queue_Space_Available;

end Queue_Package_Template;

-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

with Unchecked_Deallocation;
separate (Queue_Package_Template)

package body Allocation_Package is

   Next_Cell : List_Cell_Pointer_Type := new List_Cell_Type;

   procedure Deallocate_Cell is new
      Unchecked_Deallocation (List_Cell_Type, List_Cell_Pointer_Type);

   function New_Cell return List_Cell_Pointer_Type is

      Result : constant List_Cell_Pointer_Type := Next_Cell;

   begin  -- New_Cell

      begin
         Next_Cell := new List_Cell_Type;
      exception
         when Storage_Error => Next_Cell := null;
      end;
```

```
      return Result;

   end New_Cell;


   procedure Recycle_Cell (Cell_Pointer : in out List_Cell_Pointer_Type) is

   begin  -- Recycle_Cell

      if Next_Cell = null then
         Next_Cell := Cell_Pointer;
         Next_Cell.Link_Part := null;
         Cell_Pointer := null;
      else
         Deallocate_Cell (Cell_Pointer);
      end if;

   end Recycle_Cell;


   function Out_Of_Storage return Boolean is

   begin  -- Out_Of_Storage

      return Next_Cell = null;

   end Out_Of_Storage;


end Allocation_Package;
```

-- Solution to Exercise 10

```
with List_Package_Template;

package Unbounded_Natural_Package is

   type Unbounded_Natural is private;

   function New_Unbounded_Natural (Value : Natural) return Unbounded_Natural;
   function New_Unbounded_Natural (Value : String) return Unbounded_Natural;
   function "+" (Left, Right : Unbounded_Natural) return Unbounded_Natural;
   function "*" (Left, Right : Unbounded_Natural) return Unbounded_Natural;
   procedure Put (Item : in Unbounded_Natural);

   String_Conversion_Error : exception;

private

   subtype Digit_Subtype is Integer range 0 .. 9;
   package Digit_List_Package is new
      List_Package_Template (Element_Type => Digit_Subtype);
   type Unbounded_Natural is new Digit_List_Package.List_Type;
   -- Low-order digit is at front of list.

   -- Renaming of entities in Digit_List_Package that are not derived :

   subtype Position_Type is Digit_List_Package.Position_Type;
   function Next_Position (Position : Position_Type) return Position_Type
      renames Digit_List_Package.Next_Position;
   function Element_Value (Position : Position_Type) return Digit_Subtype
      renames Digit_List_Package.Element_Value;
   Null_Position : constant Position_Type := Digit_List_Package.Null_Position;

end Unbounded_Natural_Package;

with Text_IO; use Text_IO;

package body Unbounded_Natural_Package is


   function New_Unbounded_Natural (Value : Natural) return Unbounded_Natural is

      Result           : Unbounded_Natural;
      Remaining_Digits : Natural := Value;

   begin  -- New_Unbounded_Natural (Value : Integer) return Unbounded_Natural

      while Remaining_Digits > 0 loop
         Append_Element (Result, Remaining_Digits mod 10);
         Remaining_Digits := Remaining_Digits / 10;  --remove low-order digit
      end loop;
      return Result;

   end New_Unbounded_Natural;  -- (Value : Integer) return Unbounded_Natural
```

```
function New_Unbounded_Natural (Value : String) return Unbounded_Natural is

   Result : Unbounded_Natural_Type;

begin  -- New_Unbounded_Natural (Value : String) return Unbounded_Natural

   for I in Value'Range loop
      if Value (I) in '0' .. '9' then
         Insert_Element
            (Result,
             Character'Pos (Value (I)) - Character'Pos ('0'),
             Null_Position);
      else
         raise String_Conversion_Error;
      end if;
   end loop;

   return Result;

end New_Unbounded_Natural;  -- (Value : String) return Unbounded_Natural


function "+" (Left, Right : Unbounded_Natural) return Unbounded_Natural is

   Result         : Unbounded_Natural;
   Sum            : Integer range 0 .. 19;
   Carry          : Integer range 0 .. 1 := 0;
   Left_Position  : Position_Type := First_Position (Left);
   Right_Position : Position_Type := First_Position (Right);
   Tail_Position  : Position_Type;

begin  -- "+"

   while Left_Position /= Null_Position and
         Right_Position /= Null_Position loop

      Sum :=
         Element_Value (Left_Position) +
         Element_Value (Right_Position) +
         Carry;

      if Sum < 10 then
         Append_Element (Result, Sum);
         Carry := 0;
      else
         Append_Element (Result, Sum - 10);
         Carry := 1;
      end if;

   end loop;
```

```
      if Left_Position = Null_Position then
         Tail_Position := Right_Position;
      else
         Tail_Position := Left_Position;
      end if;

      while Tail_Position /= Null_Position loop

         Sum := Element_Value (Tail_Position) + Carry;

         if Sum < 10 then
            Append_Element (Result, Sum);
            Carry := 0;
         else
            Append_Element (Result, Sum - 10);
            Carry := 1;
         end if;

      end loop;

      if Carry = 1 then
         Append_Element (Result, 1);
      end if;

      return Result;

   end "+";


   function "*" (Left, Right : Unbounded_Natural) return Unbounded_Natural is

      Right_Digit, Carry : Digit_Subtype;
      Column_Result      : Integer range 0 .. 99;
      Result             : Unbounded_Natural;
      Left_Position      : Position_Type;
      Right_Position     : Position_Type := First_Position (Right);
      Result_Starting_Position, Current_Result_Position : Position_Type;
```

```
begin  -- "*"

   if Length (Left) = 0 or Length (Right) = 0 then
      return Result;
   end if;

   Append_Element (Result, 0);
   Result_Starting_Position := First_Position (Result);

   -- Each iteration of the outer loop starts out with Result containing
   --    a list element for every column of the partial result, except
   --    for the possible carry-out.

   loop  -- For each digit of Right, multiply all of Left by that digit.

      Right_Digit := Element_Value (Right_Position);
      Left_Position := First_Position (Left);
      Carry := 0;
      Current_Result_Position := Result_Starting_Position;

      while Left_Position /= Null_Position loop
         Product := Element_Value (Left_Position) * Right_Digit;
         Column_Result :=
            Product + Carry + Element_Value (Current_Result_Position);
         Replace_Element (Current_Result_Position, Column_Result mod 10);
         Carry := Current_Result_Position / 10;
         Left_Position := Next_Position (Left_Position);
         Current_Result_Position :=
            Next_Position (Current_Result_Position);
      end loop;

      Right_Position := Next_Position (Right_Position);

      exit when Right_Position = Null_Position;

      Append_Element (Result, Carry);  -- even if 0
      Result_Starting_Position := Next_Position (Result_Starting_Position);

   end loop;

   if Carry > 0 then
      Append_Element (Result, Carry);
   end if;

   return Result;

end "*";
```

```
procedure Put (Item : in Unbounded_Natural) is

   First_Digit_Position : Position_Type := First_Position (Item);

   procedure Put_Remaining_Digits (Position : in Position_Type) is

      package Integer_Type_IO is new Integer_IO (Integer);
      use Integer_Type_IO;

   begin  -- Put_Remaining_Digits

      if Position /= Null_Position then
         Put_Remaining_Digits (Next_Position (Position));
         Put (Element_Value (Position), Width => 1);
      end if;

   end Put_Remaining_Digits;

begin  -- Put

   if First_Digit_Position = Null_Position then
      Put ('0');
   else
      Put_Remaining_Digits (First_Position (Item));
   end if;

end Put;

end Unbounded_Natural_Package;
```

-- Solution to Exercise 11

```
generic

   type Coefficient_Type is private;

   Zero_Coefficient : in Coefficient_Type;

   with
      function "+" (Left, Right : Coefficient_Type) return Coefficient_Type
      is <>;

   with
      function "*" (Left, Right : Coefficient_Type) return Coefficient_Type
      is <>;

package Polynomial_Package_Template is

   type Polynomial_Type is private;

   function Monomial
      (Coefficient : Coefficient_Type; Power : Natural) return Polynomial_Type;
   function "+" (Left, Right : Polynomial_Type) return Polynomial_Type;
   function "*" (Left, Right : Polynomial_Type) return Polynomial_Type;

   function Zero_Polynomial return Polynomial_Type;

private

   type Term_Type is
      record
         Coefficient_Part : Coefficient_Type;
         Power_Part       : Natural;
      end record;

   package Term_List_Package is new
      List_Package_Template (Element_Type => Term_Type);

   type Polynomial_Type is new Term_List_Package.List_Type;

   -- Renaming of Term_List_Package entities that are not derived :

   subtype Position_Type is Term_List_Package.Position_Type;
   function Next_Position (Position : Position_Type) return Position_Type
      renames Term_List_Package.Next_Position;
   function Element_Value (Position : Position_Type) return Term_Type
      renames Term_List_Package.Element_Value;
   Null_Position : constant Position_Type := Term_List_Package.Null_Position;

   Null_List : Polynomial_Type renames Term_List_Package.Null_List;

end Polynomial_Package_Template;
```

```
package body Polynomial_Package_Template is

   function Zero_Polynomial return Polynomial_Type is
   begin
      return Polynomial_Type (Null_List);
   end Zero_Polynomial;

   function Monomial
      (Coefficient : Coefficient_Type; Power : Natural)
      return Polynomial_Type is

      Result : Polynomial_Type;

   begin  -- Monomial

      if Coefficient /= Zero_Coefficient then
         Append_Element (Result, Term_Type'(Coefficient, Power));
      end if;

      return Result;

   end Monomial;

   function "+" (Left, Right : Polynomial_Type) return Polynomial_Type is

      Result                  : Polynomial_Type;
      Left_Position           : Position_Type := First_Position (Left);
      Right_Position          : Position_Type := First_Position (Right);
      Leftover_Term_Position  : Position_Type;
      Coefficient_Sum         : Coefficient_Type;
      Left_Term, Right_Term   : Term_Type;

   begin  -- "+"

      while Left_Position /= Null_Position and
            Right_Position /= Null_Position loop

         Left_Term := Element_Value (Left_Position);
         Right_Term := Element_Value (Right_Position);

         if Left_Term.Power_Part < Right_Term.Power_Part then
            Append_Element (Result, Left_Term);
            Left_Position := Next_Position (Left_Position);
         elsif Left_Term.Power_Part > Right_Term.Power_Part then
            Append_Element (Result, Right_Term);
            Right_Position := Next_Position (Right_Position);
```

```
         else  -- Left_Term.Power_Part = Right_Term.Power_Part
            Coefficient_Sum :=
               Left_Term.Coefficient_Part + Right_Term.Coefficient_Part;
            if Coefficient_Sum /= Zero_Coefficient then
               Append_Element
                  (Result, Term_Type'(Coefficient_Sum, Left_Term.Power_Part));
            end if;
            Left_Position := Next_Position (Left_Position);
            Right_Position := Next_Position (Right_Position);
         end if;

      end loop;

      if Left_Position = Null_Position then
         Leftover_Term_Position := Right_Position;
      else
         Leftover_Term_Position := Left_Position;
      end if;

      while Leftover_Term_Position /= Null_Position loop
         Append_Element (Result, Element_Value (Leftover_Term_Position));
         Leftover_Term_Position := Next_Position (Leftover_Term_Position);
      end loop;

      return Result;

   end "+";


   function "*" (Left, Right : Polynomial_Type) return Polynomial_Type is

      Result, Partial_Result : Polynomial_Type;
      Left_Position          : Position_Type;
      Right_Position         : Position_Type := First_Position (Right);
      Right_Term             : Term_Type;

   begin  -- "*"

      while Right_Position /= Null_Position loop

         Left_Position := First_Position (Left);
         Right_Term := Element_Value (Right_Position);
         Partial_Result := Zero_Polynomial;

         while Left_Position /= Null_Position loop
            Left_Term := Element_Value (Left_Position);
            Append
               (Partial_Result,
                Term_Type'(Left_Term.Coefficient_Part *
                              Right_Term.Coefficient_Part,
                           Left_Term.Power_Part + Right_Term.Power_Part));
            Left_Position := Next_Position (Left_Position);
         end loop;
```

```
            Result := Result + Partial_Result;  -- polynomial addition
            Right_Position := Next_Position (Right_Position);

         end loop;

         return Result;

      end "*";

   end Polynomial_Package_Template;
```

-- Solution to Exercise 12

```
package Binary_Tree_Package is

   type Tree_Node_Type;

   type Tree_Type is access Tree_Node_Type;

   type String_Pointer_Type is access String;

   type Tree_Node_Type is
      record
         Data_Part                         : String_Pointer_Type;
         Left_Child_Part, Right_Child_Part : Tree_Type;
      end record;

end Binary_Tree_Package;


with Tree_Package; use Tree_Package;

function Reversed_Tree (Tree : Tree_Type) return Tree_Type is
begin

   if Tree = null then
      return Tree;
   else
      return
         new Tree_Node_Type'
               (Data_Part        => Tree.Data_Part,
                Left_Child_Part  => Reversed_Tree (Tree.Right_Child_Part),
                Right_Child_Part => Reversed_Tree (Tree.Left_Child_Part));
   end if;

end Reversed_Tree;
```

-- Solution to Exercise 13

```
package Tree_Package is

   type Tree_Node_Type (Number_Of_Children : Natural);

   type Tree_Type is access Tree_Node_Type;

   type Tree_List_Type is array (Positive range <>) of Tree_Type;

   type Tree_Node_Type (Number_Of_Children : Natural) is
      record
         case Number_Of_Children is
            when 0 =>
               Data_Part : Integer;
            when others =>
               Child_List_Part : Tree_List_Type (1 .. Number_Of_Children);
         end case;
      end record;

end Tree_Package;


with Tree_Package; use Tree_Package;

function Sum_Of_Leaves (Tree : Tree_Type) return Integer is

   Sum : Integer := 0;

begin  -- Sum_Of_Leaves

   if Tree.Number_Of_Children = 0 then
      return Tree.Data_Part;
   else
      for I in 1 .. Tree.Number_Of_Children loop
         Sum := Sum + Sum_Of_Leaves (Tree.Child_List_Part (I));
      end loop;
      return Sum;
   end if;

end Sum_Of_Leaves;
```

-- Solution to Exercise 14

```
with List_Package_Template;
generic

   type Element_Type is private;

   with function Has_Higher_Priority_Than
           (Element_1, Element_2 : Element_Type)
           return Boolean;

package Priority_Queue_Package is

   type Queue_Type is limited private;

   procedure Add_Element (Queue :in out Queue_Type; Element : in Element_Type);

   procedure Extract_Element
             (Queue   : in out Queue_Type;
              Highest : out Element_Type);

   function Empty (Queue : Queue_Type) return Boolean;

   Empty_Queue_Error : exception;

private

   package Queue_Package is new List_Package_Template (Element_Type);

   type Queue_Type is new Queue_Package.List_Type;

end Priority_Queue_Package;


package body Priority_Queue_Package is

   subtype Position_Type is Queue_Package.Position_Type;

   function Element_Value (Position : Position_Type) return Position_Type
      renames Queue_Package.Element_Value;

   function Next_Position (Position : Position_Type) return Position_Type
      renames Queue_Package.Next_Position;


   Null_Position : Position_Type
      renames Queue_Package.Null_Position;

   procedure Add_Element (Queue :in out Queue_Type; Element : in Element_Type) is

      Previous_Position : Position_Type := Null_Position;
      Current_Position  : Position_Type := First_Position(Queue);
      Current_Element   : Element_Type;
```

```
    begin    -- Add_Element

       while Current_Position /= Null_Position loop
          Current_Element := Element_Value (Current_Position);
          if Has_Higher_Priority_Than (Element, Current_Element) then
             Insert_Element (Queue, Element, After => Previous_Position);
             return;
          else
             Previous_Position := Current_Position;
             Current_Position := Next_Position (Current_Position);
          end if;
       end loop;

       Append_Element (Queue, Element);

    end Add_Element;


    procedure Extract_Element
                 (Queue   : in out Queue_Type;
                  Highest : out Element_Type) is

    begin

       if Length(Queue) = 0 then
          raise Empty_Queue_Error;
       else
          declare
             Position_Of_Highest : Position_Type := First_Position(Queue);
          begin
            Highest := Element_Value(Position_Of_Highest);
            Delete_Element(Queue, Position_Of_Highest);
          end; -- body
       end if;

    end Extract_Element;

    function Empty (Queue : Queue_Type) return Boolean is
    begin

       return Length(Queue) = 0;

    end Empty;

end Priority_Queue_Package;
```

```
-- Solution to Exercise 15

generic

   type Element_Type is (<>);

package Set_Package is

   type Set_Type is private;

   Empty_Set : constant Set_Type;

   type Element_List_Type is array (Positive range <>) of Element_Type;

   Extraction_Error : exception;

   function "+" (Left, Right : Set_Type) return Set_Type;

   function "*" (Left, Right : Set_Type) return Set_Type;

   function "-" (Left, Right : Set_Type) return Set_Type;

   function "-" (Set : Set_Type) return Set_Type;

   function Set_Of (Elements : Element_List_Type) return Set_Type;

   function Set_Range (Low, High : Element_Type) return Set_Type;

   function Member_Of (Element : Element_Type; Set : Set_Type) return Boolean;

   procedure Extract (From : in Set_Type; Element : in out Element_Type);

   procedure Insert (Element : in Element_Type; Into : in out Set_Type);

private

   type Set_Type is array (Element_Type) of Boolean;

   Empty_Set : constant Set_Type := (others => False);

end Set_Package;


package body Set_Package is

   function "+" (Left, Right : Set_Type) return Set_Type is
   begin
      return Left or Right;   -- union
   end "+";

   function "*" (Left, Right : Set_Type) return Set_Type is
   begin
      return Left and Right;   -- intersection
   end "*";
```

```
   function "-" (Left, Right : Set_Type) return Set_Type is
   begin
      return Left and not Right;   -- difference
   end "-";

   function "-" (Set : Set_Type) return Set_Type is
   begin
      return not Set;  -- complement
   end "-";

   function Set_Of (Elements : Element_List_Type) return Set_Type is
      Result : Set_Type := Empty_Set;
   begin
      for E in Elements'Range loop  -- constructor
         Result (Elements(E)) := True;
      end loop;
      return Result;
   end Set_Of;

   function Set_Range (Low, High : Element_Type) return Set_Type is
      Result : Set_Type := Empty_Set;
   begin
      Result (Low .. High) := (Low .. High => True);  -- constructor
      return Result;
   end Set_Range;

   function Member_Of (Element : Element_Type; Set : Set_Type) return Boolean is
   begin
      return Set (Element);  -- Membership
   end Member_Of;

   procedure Extract (From : in Set_Type; Element : in out Element_Type) is
   begin
      for E in Element_Type loop
         if From (E) then
            Element (E) := False;
            Element := E;
            return;
         end if;
      end loop;

      raise Extraction_Error;

   end Extract;

   procedure Insert (Element : in Element_Type; Into : in out Set_Type) is
   begin
      Into (Element) := True;  -- insertion
   end Insert;

end Set_Package;
```

```
          -- Solution to Exercise 16

generic

   type Element_Type is (<>);

package Set_Package is

   type Set_Type is private;

   Empty_Set : constant Set_Type;

   type Element_List_Type is array (Positive range <>) of Element_Type;

   Extraction_Error : exception;

   function "+" (Left, Right : Set_Type) return Set_Type;

   function "*" (Left, Right : Set_Type) return Set_Type;

   function "-" (Left, Right : Set_Type) return Set_Type;

   function "-" (Set : Set_Type) return Set_Type;

   function "<=" (Left, Right : Set_Type) return Set_Type;

   function Set_Of (Elements : Element_List_Type) return Set_Type;

   function Set_Range (Low, High : Element_Type) return Set_Type;

   function Member_Of (Element : Element_Type; Set : Set_Type) return Boolean;

   procedure Extract (From : in Set_Type; Element : in out Element_Type);

   procedure Insert (Element : in Element_Type; Into : in out Set_Type);

   generic
      with procedure Process_Element (Element : in Element_Type);
   procedure Process_Each_Element (Set : in Set_Type;

   generic
      with function Element_Image (Element : Element_Type) return Element_Type;
   function Set_Image (Set : Set_Type) return Set_Type;

private

   type Set_Type is array (Element_Type) of Boolean;

   Empty_Set : constant Set_Type := (others => False);

end Set_Package;
```

```
package body Set_Package is

   function "+" (Left, Right : Set_Type) return Set_Type is
   begin
      return Left or Right;   -- union
   end "+";

   function "*" (Left, Right : Set_Type) return Set_Type is
   begin
      return Left and Right;   -- intersection
   end "*";
   function "-" (Left, Right : Set_Type) return Set_Type is
   begin
      return Left and not Right;   -- difference
   end "-";

   function "-" (Set : Set_Type) return Set_Type is
   begin
      return not Set;  -- complement
   end "-";

   function "<=" (Left, Right : Set_Type) return Set_Type is
   begin
      return Left * Right = Left;   -- subset
   end "-";

   function Set_Of (Elements : Element_List_Type) return Set_Type is
      Result : Set_Type := Empty_Set;
   begin
      for E in Elements'Range loop  -- constructor
         Result (Elements(E)) := True;
      end loop;
      return Result;
   end Set_Of;

   function Set_Range (Low, High : Element_Type) return Set_Type is
      Result : Set_Type := Empty_Set;
   begin
      Result (Low .. High) := (Low .. High => True);  -- constructor
      return Result;
   end Set_Range;

   function Member_Of (Element : Element_Type; Set : Set_Type) return Boolean is
   begin
      return Set (Element);  -- Membership
   end Member_Of;
```

```
    procedure Extract (From : in Set_Type; Element : in out Element_Type) is
    begin
       for E in Element_Type loop
          if From (E) then
             Element (E) := False;
             Element := E;
             return;
          end if;
       end loop;

       raise Extraction_Error;

    end Extract;

    procedure Insert (Element : in Element_Type; Into : in out Set_Type) is
    begin
       Into (Element) := True;  -- insertion
    end Insert;

    procedure Process_Each_Element (Set : in Set_Type) is
    begin
       for E in Element_Type loop
          if Set(E) then
             Process_Element(E);
          end if;
       end loop;
    end Process_Each_Element;

    function Map_Set(Set : Set_Type) return Set_Type is
       Result : Set_Type := Empty_Set;
    begin
       for E in Element_Type loop
          if Set(E) then
             Result (Element_Image(E)) := True;
          end if;
       end loop;

       return Result;

    end Map_Set;

end Set_Package;
```

Material: Advanced Ada Topics (L305), Exercises

We would appreciate your comments on this material and would like you to complete this brief questionaire. The completed questionaire should be forwarded to the address on the back of this page. Thank you in advance for your time and effort.

1. Your name, company or affiliation, address and phone number.

2. Was the material accurate and technically correct?

   Yes ☐ No ☐

   Comments:

3. Were there any typographical errors?

   Yes ☐ No ☐

   If yes, on what pages?

4. Was the material organized and presented appropriately for your applications?

   Yes ☐ No ☐

   Comments:

5. General Comments:

place
stamp
here

COMMANDER
US ARMY MATERIEL COMMAND
ATTN: AMCDE-SB (OGLESBY)
5001 EISENHOWER AVENUE
ALEXANDRIA, VIRGINIA 22233

DTIC

END

4-86